Using CHRs to generate functional test cases for the Java Card Virtual Machine

Sandrine-Dominique Gouraud, Arnaud Gotlieb

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1725

USING CHRS TO GENERATE FUNCTIONAL TEST CASES FOR THE JAVA CARD VIRTUAL MACHINE

SANDRINE-DOMINIQUE GOURAUD AND ARNAUD GOTLIEB

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Using CHRs to generate functional test cases for the Java Card Virtual Machine

Sandrine-Dominique GOURAUD * and Arnaud GOTLIEB **

Systèmes symboliques
Projet LANDE

Publication interne n ° 1725 — Juin 2005 — 17 pages

Abstract: Functional testing based on a formal specification consists in deriving test cases from a formal model to detect faults within an implementation. In our work, we investigate the use of Constraint Handling Rules (CHRs) to automate functional test cases generation based on a formal model. Our case study is a model of the Java Card Virtual Machine (JCVM) specification written in a subset of the Coq language. In this paper we define an automated translation from this model into CHRs in order to generate test cases for the JCVM. We also propose several test purposes based on rewriting rules coverage and automatic non-conformity detection. The key point of our approach resides in the use of deep guards to model faithfully the semantic of our formal model of the JCVM. Finally, we propose an overall functional test case generation approach based on CHRs that could be applied to other formal models.

Key-words: test case generation, functional testing, constraint solving, rewriting rules, Constraint Handling Rules (CHR), Jakarta Specification Language (JSL), Java Card Virtual Machine (JCVM)

(Résumé : tsvp)

Projet RNTL CASTLES

* gouraud@irisa.fr
** gotlieb@irisa.fr

Centre National de la Recherche Scientifique (UMR 6074) Université de Rennes 1 – Insa de Rennes
Institut National de Recherche en Informatique et en Automatique – unité de recherche de Rennes

Utilisation des CHRs pour générer des cas de test fonctionnel pour la Machine Virtuelle Java Card

Résumé : Le test fonctionnel basé sur une spécification formelle consiste à dériver des cas de test à partir d'un modèle formel pour détecter des fautes dans une implémentation. Dans nos travaux, nous étudions l'utilisation des *Constraint Handling Rules* (CHRs) pour automatiser la génération de cas de test fonctionnel basée sur un modèle formel. Notre étude de cas est un modèle de la Machine Virtuelle Java Card (JCVM) écrit dans un sous ensemble du langage Coq. Dans cet article, nous définissons une traduction automatique de ce modèle sous forme de règles CHR dans le but de générer des cas de test pour la JCVM. Le point clé de notre approche réside dans l'utilisation des *deep guards* pour modéliser fidèlement la sémantique de notre modèle formel. Ensuite, nous proposons une approche globale pour la génération de cas de test fonctionnel basée sur les CHRs qui peut être appliquée à d'autres modèles formels.

Mots clés : génération de cas de test, test fonctionnel, résolution de contraintes, règles de réécriture, Constraint Handling Rules (CHR), Jakarta Specification Language (JSL), Machine Virtuelle Java Card (JCVM)

1 Introduction

Functional testing consists in 1) selecting test data from a (formal) model, 2) executing an implementation using the test set and then 3) checking the results with the help of an oracle. In functional testing, oracles provide the expected results of programs. Models that are usually used in functional testing are formal specifications such as algebraic specifications [7], B machineries [6] or finite state machines. Such formal specifications are exploited by proof tools or model checkers to prove the correctness of the implementation but sometimes, it is desirable to exploit them to generate test sets.

In our work, we are interested in functional testing based on Jakarta Specification Language (JSL) specifications. JSL [3] is a formal language based on conditional rewriting rules and it is based on a subset of the Coq language. Coq is a famous proof assistant originally designed to prove high-order theorems [1]. The JSL was defined to obtain an easy-to-read specification language which was independent of any particular tool. In the framework of RNTL CASTLES project [20] which aims at defining an environment for automating the certification of the Java Card platform [21], JSL was selected to serve as a common basis. A JSL specification of the Java Card Virtual Machine was designed [3, 4] in order to verify some security properties of the Java Card platform.

The Java Card Virtual Machine (JCVM) carries out all the instructions (or bytecodes) supported by Java Card (new, push, pop, invokestatic, invokevirtual, etc.). Our aim consists in generating test sets from the JSL specifications for each instruction, translating these abstract tests into executable tests, executing them on the implementation and eventually checking their results against the specifications. This paper addresses the former and the latter tasks which are difficult problems to deal with in Software Testing. Test concretization and test execution are not discussed here.

This paper shows how to use Constraint Handling Rules (CHRs) to generate test cases and oracles from the JSL specification of the JCVM. We define test purposes based on rewriting rules coverage and non-conformity detection and provide techniques to generate test cases that meet these objectives. Our idea is to benefit from the high declarativity of CHRs to express test purposes as well as rules of the JSL specifications into the same framework. Then, by using traditional CHR propagation and labeling, we generate tests and oracles as solutions of the underlying constraint system.

This paper is organised as follows: Section 2 introduces the syntax of JSL and its execution model; Section 3 is dedicated to the syntax and the semantic of CHRs; Section 4 introduces the translation rules which allow to translate JSL rules into CHRs; Section 5 presents two techniques to generate functional test cases according to test purposes and discuss applications to the testing of JCVM; related works are presented in Section 6, and then Section 7 ends the paper with some research perspectives.

2 The Jakarta Specification Language

The Jakarta Specification Language (JSL)[8] is a small language for describing virtual machine in a neutral mathematical style. It is a first order language with a polymorphic type system and where the functions are described by conditional rewriting rules.

2.1 Syntax

The expressions of the language JSL are first order terms with equality (==), built from term variables and from constant symbols. A constant symbol is either a constructor symbol introduced by data types definitions or a function symbol introduced by function definitions.

Let $\mathcal{C}$ be a set of constructor symbols, $\mathcal{F}$ be a set of function symbols and $\mathcal{V}$ be a set of term variables. The JSL expressions set is the term set $\mathcal{E}$ defined by $\mathcal{E} ::= \mathcal{V}|\mathcal{E} == \mathcal{E}|\mathcal{C}\mathcal{E}^*|\mathcal{F}\mathcal{E}^*$. Let *var* be the function defined on $\mathcal{E} \rightarrow \mathcal{V}^*$ which returns the set of variables of an JSL expression.

Each function symbol is defined by a set of conditional rewriting rules. This unusual format for rewriting is close to functional language with pattern-matching and proof assistant. These conditional rewriting rules are oriented and have the following form:

$$l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$$

where:

- $g = f v_1 \ldots v_m$ where $\forall i, v_i \in \mathcal{V}$ and $\forall i, j, v_i \neq v_j$
- l_i is either a variable or a function which does not introduce new variables:
  for $1 \leq i \leq n, var(l_i) \subseteq var(g) \cup var(r_1) \cup \ldots \cup var(r_{i-1})$
- r_i should be a value called *pattern* (built from variables and constructors), should contain only fresh variables and should be linear[1]:
  for $1 \leq i, j \leq n$ and $i \neq j$, $var(r_i) \cap var(g) = \emptyset$ and $var(r_i) \cap var(r_j) = \emptyset$
- d is an expression and $var(d) \subseteq var(g) \cup var(r_1) \ldots \cup var(r_n)$

The rule means if for all i, l_i can be rewritten into r_i then g is rewritten into d. Thereafter, these rules are called JSL rules.

Example 1 gives a JSL definition, extracted from the JCVM specification, of the function *plus*.

Example 1 (JSL rules to define *plus*)
data nat $=$ $0 \mid S\, nat$.
function **plus** :=
$\langle plus\_r1 \rangle \quad n \rightarrow 0 \quad \Rightarrow \quad (plus\, n\, m) \rightarrow m;$
$\langle plus\_r2 \rangle \quad n \rightarrow (S\, p) \quad \Rightarrow \quad (plus\, n\, m) \rightarrow (S\, (plus\, p\, m)).$

Where n, m and p are some variables, 0 are S are some constructor symbols and plus *is a function symbol. The construction $\langle \ldots \rangle$ allow to give a name to a rule: the first rule of plus is referred as $\langle plus\_r1 \rangle$ and the second one as $\langle plus\_r2 \rangle$.*

Partial functions and non-deterministic functions can be defined.

[1] All the variables should be different: the term $C\, v\, v$ where $v \in \mathcal{V}$ and $C \in \mathcal{C}$ is not allow.

2.2 Execution model

Given a term e, we recall that each subterm of e can be identified by a position p, $e_{|p}$ is the subterm of e at position p. The expression $e[p \leftarrow d]$ means that in the term e, the subterm at the position p is replaced by the term d. A substitution θ can be seen as a renaming of variables.

Let $\mathcal{R}$ a set of rewriting rules. An expression e is rewritten into an expression e' if there exists a rule in $\mathcal{R}$, $l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$, a position p in e and a substitution θ such as:

- $e_{|p} = \theta g$ and $e' = e[p \leftarrow \theta d]$
- for $1 \leq i \leq n, \theta l_i \rightarrow^* \theta r_i$ where $\rightarrow^*$ is the reflexive and transitive cloture of $\rightarrow$

Example 2
Given the JSL definition of plus*, the term* $(plus\ 0\ (plus(S\ 0)\ 0))$ *can be rewritten into* $(S\ 0)$ *in three derivations:*

$$\underline{(plus\ 0\ (plus\ (S\ 0)\ 0))} \rightarrow_{r1} (plus\ (S\ 0)\ 0) \rightarrow_{r2} (S\ (plus\ 0\ 0)) \rightarrow_{r1} (S\ 0)$$

$$(plus\ 0\ \underline{(plus\ (S\ 0)\ 0)}) \rightarrow_{r2} \underline{(plus\ 0\ (S\ (plus\ 0\ 0)))} \rightarrow_{r1} (S\ (plus\ 0\ 0)) \rightarrow_{r1} (S\ 0)$$

$$(plus\ 0\ \underline{(plus\ (S\ 0)\ 0)}) \rightarrow_{r2} (plus\ 0\ (S\ \underline{(plus\ 0\ 0)})) \rightarrow_{r1} (plus\ 0\ (S\ 0)) \rightarrow_{r1} (S\ 0)$$

3 Background on Constraint Handling Rules

This section is inspired of Thom Frühwirth's survey [12], the book [13] and the dedicated website [22]. The Constraint Handling Rules (CHR) language is a committed-choice language, which consists of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. This language extends a host language with constraint solving capabilities. Implementations of CHRs are available in Eclipse Prolog, Sicstus Prolog, HAL, etc.

CHR are based on two principles: **Simplification** where constraints are replaced by more simpler ones while logical equivalence is preserved and **Propagation** where new constraints which are logically redundant are added to cause further simplification.

In this language, a constraint is a special first-order predicate. This predicate is either a built-in (predefined) constraint which already exists in the host language or a CHR (user-defined) constraint which is defined by a CHR program (finite set of CHR rules).

3.1 Syntax

There are three kinds of CHR rule:

- Simplification CHR rules are of the form `H <=> G | B`
- Propagation CHR rules are of the form `H ==> G | B`
- Simpagation CHR rules are of the form `H\H' <=> G | B`

More precisely, the multi-head H and H' are non-empty conjunctions of CHR constraints, the guard G is a conjunction of built-in constraints and the body B is a conjunction of built-in and CHR constraints.

In this section, we will focus on simplification and propagation rules which correspond *explicitly* to the two principles on which CHR are based.

Example 3 (`plus` defined with CHR rules)

```
R1@ plus(A,B,R)  <=> A=0     | R=B.
R2@ plus(A,B,R)  <=> A=s(C)  | plus(C,B,D), R=s(D).
 C@ plus(A,B,R)  ==> plus(B,A,R).
```

The construction `...@` *gives a name to a CHR rule: the first simplification rule of* `plus` *is referred as* `R1`*, the second one as* `R2` *and the propagation rule as* `C`*.*

3.2 Semantic

Given a constraint theory (CT) (with true, false and an equality constraint $=$) which determines the meaning of the built-in constraints, the declarative interpretation of a CHR program is given by a conjunction of universally quantified logical formula. There is a formula for each rule.

If $\bar{x}$ denotes the variables occurring in the head H and $\bar{y}$ (resp. $\bar{z}$) the variables occurring in the guard (resp. body) of the rule, then

- a simplification CHR rule is a logical equivalence if the guard is satisfied:

$$\forall \bar{x}(\exists \bar{y} G \rightarrow (H \leftrightarrow \exists \bar{z} B))$$

- a propagation CHR rule is an implication if the guard is satisfied:

$$\forall \bar{x}(\exists \bar{y} G \rightarrow (H \rightarrow \exists \bar{z} B))$$

The operational semantic of CHR programs is given by a transition system where a state is a conjunction of constraints. Initial state consists of the goal and final states are either successful (the conjunction is consistent) or failed (the conjunction is inconsistent). There are one transition for solving built-in constraints (Solve) and three transitions for applying each kind of CHR (Simplify, Propagate and Simpagation).

Solve	If	C is a built-in constraint
	And	$CT \models (C \wedge D) \leftrightarrow D'$
	Then	`C,D` $\mapsto$ `D'`
Simplify	If	`H <=> G | B`
	And	$CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$
	Then	`H',D` $\mapsto$ `D,H=H',B`
Propagate	If	`H ==> G | B`
	And	$CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$
	Then	`H',D` $\mapsto$ `H',D,H=H',B`

Rules are applied fairly (every rule that is applicable is applied eventually). Moreover, propagation rule is applied at most once on the same constraints in order to avoid trivial non-termination. CHR programs can be non-confluent and if guards of rules for a same predicate are not mutually exclusive then they can be non-determinate programs.

Example 4 (Simple uses of CHRs)
These two examples shows how CHRs work.

```
        plus(s(0),s(0),R)
↦R2     plus(0,s(0),R1), R=s(R1)
↦R1     R1=s(0), R=s(R1)
↦Solve  R=s(s(0))
```

This second example exploits the propagation rule: without this rule, the term `plus(M,s(0),s(s(0)))` *was delayed.*

```
        plus(M,s(0),s(s(0)))
↦C      plus(M,s(0),s(s(0))), plus(s(0),M,s(s(0)))
↦R2     plus(M,s(0),s(s(0))), plus(0,M,s(0))
↦R1     plus(M,s(0),s(s(0))), M=s(0)
↦Solve  plus(s(0),s(0),s(s(0))), M=s(0)
↦R2     plus(0,s(0),s(0)), M=s(0)
↦R1     s(0)=s(0), M=s(0)
↦Solve  M=s(0)
```

The CHR language was designed for writing efficient and generalised constraint solvers [12]. The following example shows one of the main interest of CHR.

Example 5
Here, the relation $M = N$ is deduced whereas this relation wouldn't be obtained with a traditional Herbrand solver.

```
        plus(M,0,N)
↦C      plus(M,0,N), plus(0,M,N)
↦R1     plus(M,0,N), M=N
↦Solve  plus(M,0,M), M=N
```

4 JSL to CHR translation rules

The first task of our approach is based on an automatic translation of JSL specifications into CHRs. The translation rules are given below under the form of judgments.

4.1 Translation rules

There are three kinds of translation rules: translation rules for expressions, translation rules for rewriting rules (main operator $\rightarrow$) and translation rules for JSL rules (main operator $\Rightarrow$). In the following judgments, `r` denotes a fresh variable.

The judgment $e \rightsquigarrow \mathtt{t} \triangleleft \{\mathtt{C}\}$ states that JSL expression e is translated into term $\mathtt{t}$ under the conjunction of constraints $\mathtt{C}$. Term $\mathtt{t}$ is a variable, an atom or a CHR constraint.

$$\frac{}{v \rightsquigarrow \mathtt{v} \triangleleft \{\mathtt{true}\}} \qquad \frac{}{c \rightsquigarrow \mathtt{c} \triangleleft \{\mathtt{true}\}}$$

$$\frac{e_1 \rightsquigarrow \mathtt{t_1} \triangleleft \{\mathtt{c_1}\} \ \ldots \ e_n \rightsquigarrow \mathtt{t_n} \triangleleft \{\mathtt{c_n}\}}{c\, e_1 \ldots e_n \rightsquigarrow \mathtt{c}(\mathtt{t_1}, \ldots, \mathtt{t_n}) \triangleleft \{\mathtt{c_1}, \ldots, \mathtt{c_n}\}}$$

$$\frac{e_1 \rightsquigarrow \mathtt{t_1} \triangleleft \{\mathtt{c_1}\} \ \ldots \ e_n \rightsquigarrow \mathtt{t_n} \triangleleft \{\mathtt{c_n}\}}{f\, e_1 \ldots e_n \rightsquigarrow \mathtt{r} \triangleleft \{\mathtt{c_1}, \ldots, \mathtt{c_n}, \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n}, \mathtt{r})\}}$$

The judgment $e \rightarrow p \rightsquigarrow \mathtt{C}$ states that if a JSL expression e is rewritten into a pattern p then this rule is translated into the conjunction of constraints $\mathtt{C}$. If e is a variable, $\mathtt{C}$ is reduced to an equality $\mathtt{e{=}p}$ into two terms. If e is a function call, $\mathtt{C}$ contains the constraints which constraint the different arguments plus the CHR constraint associated to the function.

$$\frac{}{v \rightarrow p \rightsquigarrow \mathtt{v} = \mathtt{p}}$$

$$\frac{e_1 \rightsquigarrow \mathtt{t_1} \triangleleft \{\mathtt{c_1}\} \ \ldots \ e_n \rightsquigarrow \mathtt{t_n} \triangleleft \{\mathtt{c_n}\} \quad p \rightsquigarrow \mathtt{p} \triangleleft \{\mathtt{true}\}}{f\, e_1 \ldots e_n \rightarrow p \rightsquigarrow \mathtt{c_1}, \ldots, \mathtt{c_n}, \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n}, \mathtt{p})}$$

The judgment $l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d \rightsquigarrow \mathtt{g'} \Leftrightarrow \mathtt{guards}|\mathtt{body}$ states that the JSL rule $l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$ is translated into the CHR rule $\mathtt{g'} \Leftrightarrow \mathtt{guards}|\mathtt{body}$ where $\mathtt{g'}$ is a CHR constraint associated to the expression g, $\mathtt{guards}$ is the conjunction of constraints corresponding to the translation of the rules $l_i \rightarrow r_i$, and $\mathtt{body}$ is a conjunction of constraints corresponding to the translation of the expression d.

$$\frac{l_1 \rightarrow r_1 \rightsquigarrow \mathtt{g_1} \quad \ldots \quad l_n \rightarrow r_n \rightsquigarrow \mathtt{g_n} \qquad\qquad e \rightsquigarrow \mathtt{t} \triangleleft \{\mathtt{B}\}}{l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow f\, v_1 \ldots v_k \rightarrow e \ \rightsquigarrow \ \mathtt{f}(\mathtt{v_1}, \ldots, \mathtt{v_k}, \mathtt{r}) \Leftrightarrow \mathtt{g_1}, \ldots, \mathtt{g_n}|\mathtt{B}, \mathtt{r} = \mathtt{t}.}$$

It is important to note that non–determinism, confluence and recursivity are preserved by the translation.

4.2 Deep Guards

In our translation, we considered that CHR's guards could be made of prolog goals and CHR calls. This approach, which is referred to as deep guards, has received much attention by the past. For example, Smolka recalls in [17] that "deep guards constitute the central mechanism to combine processes and (encapsulated) search for problem-solving". Deep guards are used in several systems such as AKL, Oz [19] or HAL [10]. Deep guards rely on how guard entailment is tested when the guards are not only built-in predicates but also constraint calls. Technically, a guard entailment test is called "ask constraint" whereas a constraint added to the constraint store is called "tell constraint" and both operations are clearly distinct. For example, if the constraint store contains $Z = f(X, Y), Y = a$ then a tell constraint $X = Y$ where $=$ denotes Prolog's unification, will result in the store $Z = f(a, a)$ whereas the corresponding ask constraint will just suspend until it would be entailed or disentailed. Note that each time a CHR clause is woken, its guard either succeeds or fails. If the guard succeeds, one commits to it and then the body is executed.

The current approach[2] to deal with deep guards that contain Prolog goals (but not CHR calls) consists in considering guards as tell constraints and checking at runtime that no guard variable is modified. This approach is based on the fact that the only way of constraining terms in the Herbrand Universe is unification. Note that the ask constraint that corresponds to the unification in the Herbrand Universe is well-known: it is the term equality test. For example, if $X = Y$ is the tell constraint then $X == Y$ corresponds to its ask constraint. Note also this approach is no longer efficient when we consider long term computations as guards are executed every time a CHR clause is woken. An approach for this problem consists in precomputing the guard by executing the Prolog goal only once, and testing entailment on the arguments of the clause. When Prolog goals are involved into the guards, the guard entailment test is no more decidable as non-terminating computations can arise. Fortunately, in our case, this cannot happen as the CHR's guards are obtained by an automatic translation from the JSL specifications of the JCVM [8] which is only made of terminating rules.

When CHRs are involved into the guards, the problem is more difficult as guards themselves can set up constraints. In that case, considering guards as tell constraints is no longer correct as wrong deductions can be made. Our approach for this problem consists in suspending the guard entailment test until it becomes decidable. More precisely, the guard entailment test is delayed until all the guard variables become instantiated[3]. At worst, this instantiation arises during the labeling process. Of course, this approach leads to fewer deductions at propagation time but we believe that it remains acceptable to deal with deep guards that contain CHRs.

4.3 Implementation

All the translation rules were implemented into the library `JSL2CHR.pl`. Given a file containing JSL definitions, the library builds an abstract syntax tree by using a Definite Clause Grammar of JSL, and then produces automatically the CHR rules.

The library was used on the JSL specifications of the JCVM, which is composed of 310 functions. As a result, 1537 CHR rules were generated.

5 Tests generation for JCVM

Let us recall that in the frame of the RNTL CASTLES project, our work consists in defining techniques to generate functional test cases for an implementation of the JCVM. Based on the semantic-preserving translation described above, our starting point is a set of CHRs that can be considered as a formal specification. This section is devoted to how generate test data and oracles for testing an implementation of the JCVM against this formal model. Functional testing relies on test purposes. In this paper, we discuss of two possible test purposes that lead to two distinct techniques.

In order to directly illustrate our approach on real JSL rules, we introduce the JCVM in Section 5.1, then we present the two approaches that we proposed: the first one (Section 5.2) is based on the structural coverage of JSL/CHR rules whereas the second one (Section 5.3) is based on security policies that must be satisfied by any implementation of the JCVM.

[2]The approach which is followed in many implementation of CHRs such as Sicstus Prolog, Eclipse Prolog, HAL [10]

[3]Naturally this solution is close to the traditional techniques of coroutinage in Prolog as implemented by `freeze` and `delay`.

5.1 The Java Card Virtual Machine

We consider a specification of the JCVM that is automatically derived from a Coq formalisation previously done in the Certificates context [5]. In this formalisation, the JCVM works like a state machine and is described by small-step semantics: each instruction is formalised as a state transformer. The inputs of the virtual machine are the initial state and a program. The output is a new state obtained after the execution of the code associated of the program. Each state contains all the elements manipulated by a program during its execution: values, objects and an execution environment for each called method. States are formalised as record consisting of a heap (he) which contains the objects created during execution, a static heap (sh) which contains static fields of classes and a stack of frames (fr) which are the environments for executing methods. States are also tagged with a label Normal if the execution is correct or Abnormal if an exception (or an error) is raised. The byte code instruction *push* and the function *stack_f* in the example 6 are directly extracted from the JSL specification of the JCVM.

Example 6 (The JSL specification of push)
Given a JCVM state, the function **stack_f** *returns the stack of frames fr (environments for executing methods) of this state.*

$function$ **stack_f** :=
$\langle stack\_f\_r1 \rangle \quad st \rightarrow (Build\_jcvm\_state\ sh\ he\ fr) \Rightarrow (stack\_f\ st) \rightarrow fr.$

Given a primary type, a value and a JCVM state, if the stack of the state st is empty then the function **push** *terminates on an error else the stack of the execution method environments of st is updated using the function update_frame: in particular, the operand stack of the first execution method environment h is updated adding the value x of type t, this update is done by the res function.*

$function$ **push** :=
$\langle push\_r1 \rangle \quad (stack\_f\ st) \rightarrow Nil \qquad \Rightarrow (push\ t\ x\ st) \rightarrow (abortCode\ State\_error\ st);$
$\langle push\_r2 \rangle \quad (stack\_f\ st) \rightarrow (Cons\ h\ lf) \quad \Rightarrow (push\ t\ x\ st) \rightarrow (update\_frame(res\ t\ x\ h)\ st).$

Table 5.1 gives the CHR rules automatically produced by our library from the JSL specifications given in the example 6.

```
stack_f_r1  @ stack_f(St,R)    <=> St=build_jcvm_state(Sh,He,Fr)
                               | R=Fr.
push_r1     @ push(T,X,St,R)   <=> stack_f(St,nil)
                               | abortCode(state_error(St),Ra), R=Ra.
push_r2     @ push(T,X,St,R)   <=> stack_f(St,cons(H,Lf))
                               | res(T,X,H,Res), update_frame(Res,St,Ru), R=Ru.
```

Table 1: CHR rules for push

5.2 A rule, a test

Our first approach is inspired of classical functional testing techniques which consist in designing test cases in order to test each rule of the specification. These techniques are based on two assumptions: the correctness of the specification and the uniformity hypothesis[7]. As usual, the specification is considered to be a reference and then it is assumed to be correct. The uniformity hypothesis says that if a rule has a correct behavior for a single test case then it has a correct behavior for all test cases that sensitive the rule. Of course, this assumption is very strong and nothing can prevent it to be violated but this is an usual assumption in functional testing. Recall that testing just looks for flaws within an implementation and does not try to prove correctness.

5.2.1 Overview

For each JSL specification defining a function, our goal is to find a test case (a substitution of the variables) which activate each rules of the specification. To achieve this task, a search process over the possible terms has to be performed. For example, consider the problem of covering all the JSL rules which define the instruction *push*. To activate the *push_r1* rule, the state's stack *st* must be rewritten into Nil (i.e. to be empty) whereas to activate the *push_r2* rule, the state's stack *st* must be rewritten into $Cons\ h\ lt$ (i.e. to have at least one frame). Note that no constraints hold over t and x and in this case, a random labeling process is called to instantiate the variables. Hence, to ensure complete coverage of the *push* instruction, we must generate (t, x, st) where *st* such as:

- *st* allows this rewriting $stack\_f\ st \rightarrow Nil$
- *st* allows this rewriting $stack\_f\ st \rightarrow Cons\ h\ lt$

For example, our process must be able to generate the following solutions[4]:

$(Boolean, POS(XI(XO(XH))), Build\_jcvm\_state(Nil, Nil, Nil))$
and
$(Byte, NEG(XH), Build\_jcvm\_state(Nil, Nil, Cons(Build\_frame(Nil, Nil, S(S(0)),$
$Build\_Package(0, S(0), Nil), True, S(0)), Nil)))$

To automatically generate these inputs, we use the CHR rules associated to the JSL rules and the classical techniques of constraint solving (labeling and backtracking process) in order to unify input variables like *st* to the required terms. Cover a JSL rule consists in finding inputs such as the guard of the corresponding CHR rule is satisfied.

As usual in constraint programming, constraints play here an active role as relations are considered before labeling (test-and-generate approach). Note that this solution contrasts with classical functional testing techniques that usually instantiate first the variables and then check if they satisfy the guards (generate-and-test approach).

[4] $Build\_jcvm\_state$, $Build\_frame$, $Build\_Package$, XI, $X0$, XH, POS, $Byte$, NEG, $Boolean$ and $True$ are constructor symbols defined by the JSL specification JSL of the JCVM.

5.2.2 Constraints

Moreover setting up the constraints before launching the labeling process is more complex as it appears in our case, as deep guards are accepted. When a guard that contains CHR and Prolog constraints is selected, satisfaction of the nested guards have to be considered too. Indeed, consider any CHR rule $\mathtt{r} : \mathtt{H} \Leftrightarrow \mathtt{G}|\mathtt{B}$ where $\mathtt{G}$ is $\mathtt{p_1}, \ldots, \mathtt{p_n}$. Satisfy the guard $\mathtt{G}$ consists in satisfying at least one guard of CHR rules defining each predicate $\mathtt{p_i}$ which appears in $\mathtt{G}$ i.e. in finding a valuation of the variables such as the constraint is simplified either in `true` or in a consistent conjunction of equalities. Simplification process need that at least one of the guards of CHR rules defining the constraint $\mathtt{p_i}$ be satisfied. Test purpose for each CHR rule is expressed by the following formula:

$$\bigwedge_i \left(\bigvee_j guard(\mathtt{p_i}, j) \right)$$

where $\mathtt{p_i}$ is a CHR predicate appearing in $\mathtt{G}$ and $guard(\mathtt{p_i}, j)$ is the guard of the jth rules defining $\mathtt{p_i}$. Any solution of the formula is a test data which activate the CHR rule $\mathtt{r}$.

To find a solution of this conjunction, we consider the following search tree:

- each node is a state where solution searching process can arrive: it is a conjunction of CHR and Prolog constraints;
- each leaf is a conjunction of constraints without any CHR constraint.

Given a node and a CHR constraint $\mathtt{p}$ appearing in this node, each branch b_i correspond to the substitution of $\mathtt{p}$ by the $guard(\mathtt{p}, i)$.

Note that such tree can be infinite. As we are looking for one solution and not all solutions (exhaustivity), we can prune some branches. Moreover, there is as many search trees as constraint shedulings and CHR rules. In our case, as we looking for one solution, the efficiency of the solution search process can be improved using heuristics which guide the choice of the next branch to be explored. Our first heuristic consists in dealing with CHR constraints in a node using the following order (noted ¡):

1. Choose the CHR constraint defined by less rules;
2. If two constraints have the same number of rules, use the lexical order.

With a such heuristic, we first choose to replace `stack_f`, then `plus` and finally `push`. That give us `stack_f < plus < push`.

In the search tree, if $\mathtt{p_i} < \mathtt{p_j}$ then $\mathtt{p_i}$ should not appear in a subtree which root is a conjunction containing $\mathtt{p_j}$ and not $\mathtt{p_i}$[5]. If a node is a conjunction which has not any solution or contains a CHR constraint which can not be reduced, then the node is a leaf labeled with `fail`. A solution is a leaf which is not labeled with `fail`.

[5]This is to avoid infinite branches resulting from recursive or mutually recursive definitions.

Example 7 *Consider the following CHR rules:*

```
foo(A,B,L,R)  <=> foo2(A,L,R2), foo1(A,B,R1)  | ...
foo1(A,B,R)   <=> A=0                         | ...
foo1(A,B,R)   <=> A=s(C)                      | ...
foo2(A,L,R)   <=> L=nil                       | ...
foo2(A,L,R)   <=> A=s(B), L=cons(B,nil)       | ...
foo2(A,L,R)   <=> foo3(L,R1)                  | ...
foo3(L,R)     <=> L=nil                       | ...
foo3(L,R)     <=> L=cons(A,L2), foo1(A,A,R)   | ...
```

Test cases, to cover the rule defining foo, are the valid leaves (see Fig.1).

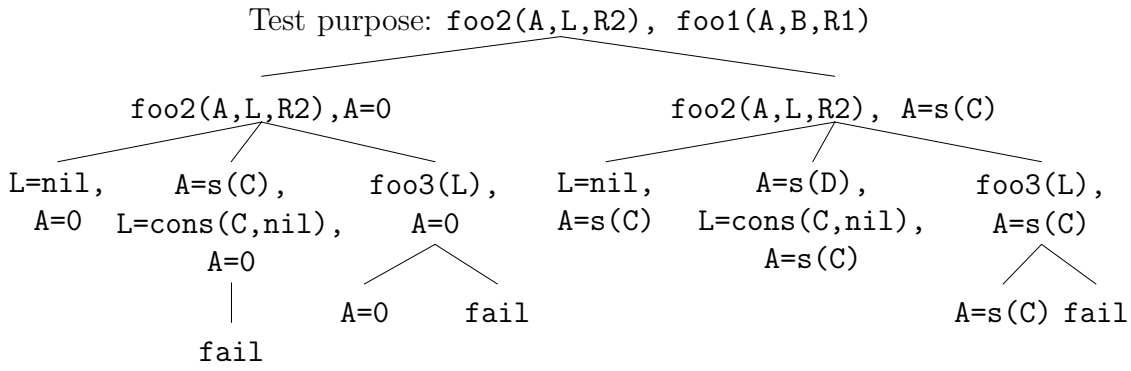

Figure 1: Search tree for test purpose `foo2(A,L,R2), foo1(A,B,R1)`

5.2.3 The labeling process

The labeling process consists in trying to instantiate each free variables of a term. This process can be based on deterministic or randomised [15] heuristics. In software testing approaches, random selection is usually preferred as it improves the flaws detection capacity. Randomised heuristics are based on random choices based on probabilistic distributions. The simplest approach consists in generating terms based on uniform distribution. Lot of works have been carried out to address the problem of uniform generation of terms and are related to the random generation of combinatorial structures [11]. Our previous work on that subject is of a great help in this process [9] as we defined techniques and tools to generate randomly combinatorial structures in the context of statistical software testing.

In this first approach, we showed how the CHRs and constraint solving techniques can be used to generate test cases in order to cover each rule of a JSL definition of a JCVM bytecode instruction. The rule to be tested and the test purpose are expressed by constraints.

5.3 Conformance testing

Our second approach is based on the notion of conformance testing which consists in testing whether an implementation satisfy a given security policy P. The property is a test purpose for test case selection and the goal is to find test cases that reject the implementation. In this kind of approaches, both the specification and the property are assumed to be correct. Afterward, we suppose that the property is specified by JSL rules.

Given a property *prop*, let $\mathcal{F}(prop)$ be the set of functions/predicates targeted by the property *prop*. The idea is to generate test cases covering the JSL rules of a given property *prop* by exploiting the JSL specifications of the functions/instructions which are in $\mathcal{F}(prop)$. We base our method on a *projection* process which crosses a property with the targeted specifications in order to refine the test purpose. This projection process consists in replacing the CHR rules of *prop* by enhanced rules until to obtain a decomposition of initial test purpose into several more precise test purposes.

The main operator of the projection process is a function *proj* which given two CHR rules $\mathtt{r_1}$ and $\mathtt{r_2}$ and a CHR predicate `p` defined by $\mathtt{r_2}$, enhances the rule $\mathtt{r_1}$ with the guards of r_2 and replaces the predicate `p` in the body of $\mathtt{r_1}$ by the body of $\mathtt{r_2}$:

If	$\mathtt{r_1}$ is of the form `H <=> G'| p,B'`
And	$\mathtt{r_2}$ is of the form `q <=> G|B`
And	$\exists\theta$ is such as $\theta\mathtt{q} = \mathtt{p}$
Then	$proj(\mathtt{r_1}, \mathtt{r_2}, p)$ is `H <=> G',`θ`G| `θ`B,B'`

Let $\mathcal{P}(\mathtt{r}, prop)$ be the set of CHR predicates which appear in the body of `r` and define a function/instruction of $\mathcal{F}(prop)$, and let $\mathcal{R}(\mathtt{p})$ be the set of CHR rules defining `p`. The projection process is defined by the following algorithm:

Given a rule `r` defining the property *prop*
  $E = \{\mathtt{r}\}$.
  For all `p` in $\mathcal{P}(\mathtt{r}, prop)$
    $NewE = \emptyset$
    For all $\mathtt{r_e}$ in E
      For all $\mathtt{r_p}$ in $\mathcal{R}(\mathtt{p})$
        $NewE = NewE \cup \{proj(\mathtt{r_e}, \mathtt{r_p}, p)\}$
    $E = NewE$

Note that all rules which guards cannot be satisfied are removed. The guards of all the rules appearing in E correspond to test cases to activate the rule `r`. This process is applied to each rule defining the property *prop*.

Let us illustrate this process on the case of associativity property of the function *plus*.

```
asso_r1 @  asso(A,B,C,R)  <=> plus(B,C,R1), plus(A,R1,R2), plus(A,B,R3),
                              plus(R3,C,R4), eq(R2,R4,R).
plus_r1 @  plus(A,B,R)    <=> A=0             | R=B.
plus_r2 @  plus(A,B,R)    <=> A=s(C)          | plus(C,B,R1), R=R1(C).
eq_r1 @    eq(N,M,R)      <=> N=0, M=0        | R=true.
eq_r2 @    eq(N,M,R)      <=> N=0, M=s(P)     | R=false.
eq_r3 @    eq(N,M,R)      <=> N=s(P), M=0     | R=false.
eq_r4 @    eq(N,M,R)      <=> N=s(P), M=s(Q)  | eq(P,Q,R).
```

Selecting test cases from the associativity property of the function *plus* leads to instantiate the tuple (`A`, `B`, `C`). After apply the algorithm, the initial rule `asso_r1` is projected into four new rules:

```
asso(A,B,C,R)  <=> B=0, A=0
               | R1=C, R1=R2, R3=B, plus(R3,C,R4), eq(R2,R4,R).
asso(A,B,C,R)  <=> B=0, A=s(D)
               | R1=C, R2=s(R5), R3=s(R6), plus(D,R1,R5), plus(D,B,R6),
                 plus(R3,C,R4), eq(R2,R4,R).
asso(A,B,C,R)  <=> B=s(D), A=0
               | R1=s(R5), R1=R2, B=R3, plus(D,C,R5), plus(R3,C,R4),
                 eq(R2,R4,R).
asso(A,B,C,R)  <=> B=s(D), A=s(E)
               | R1=s(R5), R2=s(R6), R3=s(R7), plus(D,C,R5), plus(E,R1,R6),
                 plus(E,B,R7), plus(R3,C,R4), eq(R2,R4,R).
```

The predicate `plus(R3,C,R4)`$\in \mathcal{P}$(`asso_r1`, *asso*) will be not exploited because none constraint on `A`, `B` or `C` can be added.

Once the initial test purpose is decomposed several more precise test purposes, it remains to generate a test data for each new rule by using for example, the approach presented in Section 5.2.

Although, we did not yet apply this principle on the JSL specification of the JCVM, we give here the flavor of the approach on a realistic security property. The following property (labeled FDP_RIP.1) is extracted from the Sun document[2] which defines the security requirements for the platform Java Card.

Property: Protection of residual information
The platform should assure that each information which was contained in a memory-resource is not available when this memory-resource is allocated to new objects like class instances and arrays. When such object is created, the fields of the object and the elements of the array are initialised with default values.

Classically, properties are given in a natural language, and a first non-trivial step consists in translating these properties into a formal language.

6 Related Work

Bernot and al. [7] pioneered the use of Logic Programming to construct test sets from formal specifications. Starting from an algebraic specification, test sets were selected using equational logic programming (Horn clause logic). A few years ago, Constraint Logic Programming was explored by Gotlieb and al. [14] to generate test sets for structural testing of C programs. Given the source code of a program, a semantically-equivalent constraint logic program was built and questioned to find test data that cover a given testing criterion. Legeard and al.[6] proposed a method for functional boundary testing from B and Z formal specifications based on Set constraint solving techniques (CLP($\mathcal{S}$)) and apply it to the transaction mechanism testing of the Java Card.

However, in these works, CHRs were not explored for test case generation. On the contrary, Lötzbeyer and Pretschner [18, 16] proposed a software testing technique that use CHR solving. Models were finite state automata describing the behavior of the system under test and test

cases were sequence of input/output events. CHR were used to define new constraint solvers allowing so the generation complex data types.

Our work distinguishes by the systematic translation of formal specifications into CHRs, by making use of deep guards. Our approach does not restrict the form of guards and appear as being more declarative to generate test cases.

7 Conclusion

In this paper, we have proposed to use the CHRs to generate functional test cases for a JCVM implementation. A JSL specification of the JCVM has been used as a formal model and test generation has been designed to satisfy two test purposes: structural coverage of CHR rules and non-conformity detection against a security property.

This ongoing work seems to be promising: translating a JSL specification into equivalent CHRs is possible and lead to automatic test case generation. It remains to write a formal proof of the correctness of the translation and to generate a complete test set for the JCVM.

At the testing level, we need to formalise the projection process in order to describe new testing criteria based on formal models. We also need to study how to improve the implementation of deep guards.

Further, we believe the approach could be easily extended to deal with other formal models. CHRs appear as being an efficient tool to generate test cases.

8 Acknowledgments

We wish to acknowledge E. Coquery for fruitful discussion on CHR, and G. Dufay and G. Barthe who give us all informations on JSL that we need. This work is supported by the RNTL Project CASTLES.

References

[1] The Coq proof assistant. `http://coq.inria.fr/`.

[2] Java Card System Protection Profile Collection (version 1.0b), 2003. `http://java.sun.com/products/javacard/pp.html`.

[3] G. Barthe, G. Dufay, M. Huisman, and S. Sousa. Jakarta: a toolset for reasoning about JavaCard. In *Proceedings of E-smart 2001*, volume 2140 of *LNCS*, pages 2–18. In I. Attali and T. Jensen Eds, Springer-Verlag, 2001.

[4] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In *Proceedings of ESOP'01*, volume 2028 of *LNCS*, pages 302–319. D. Sands Eds, Springer-Verlag, 2001.

[5] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. Melo de Sousa, and S-W. Yu. Formalization of the JavaCard Virtual Machine in Coq. In *Proceedings of FTfJP'00 (ECOOP Workshop on Formal Techniques for Java Programs)*, pages 50–56. S. Drossopoulou and al, Eds, 2000.

[6] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *International Journal of Software Practice and Experience*, 34(10):915–948, 2004.

[7] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[8] Simo Melo de Sousa. *Outils et techniques pour la vérification formelle de la plate-forme JavaCard*. PhD thesis, Université de Nice, février 2003.

[9] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A Generic Method for Statistical Testing. In *Fifteenth IEEE International Symposium on Software Reliability Engineering*, pages 25–34, 2004.

[10] G.J. Duck, P.J. Stuckey, M. Garcia de la Banda, and C. Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming (PPDP03)*, page 2003, 79-90.

[11] Ph. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132:1–35, 1994.

[12] T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Logic Programming*, 37(1-3), october 1998. Special Issue on Constraint Logic Programming, In P. Stuckey and K. Marriott Eds.

[13] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Cognitive Technologies. Springer Verlag, 2003. ISBN 3-540-67623-6.

[14] A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *Constraints Stream, First International Conference on Computational Logic*, number 1891 in LNAI, pages 399–413. Springer-Verlag, 2000.

[15] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A New Way of Automating Statistical Testing Methods. In *Sixteenth IEEE International Conference on Automated Software Engineering (ASE)*, pages 5–12, 2001.

[16] H. Ltzbeyer and A. Pretschner. AutoFocus on Constraint Logic Programming. In *Proceedings of (Constraint) Logic Programming and Software Engineering (LPSE'2000)*, july 2000.

[17] A. Podelski and G. Smolka. Situated Simplification. In *Proceedings of the 1st Conference on Principles and Practice of Constraint Programming*, volume 976 of *LNCS*. Springer-Verlag, 1995.

[18] A. Pretschner and H. Ltzbeyer. Model Based Testing with Constraint Logic Programming: First Results and Challenges. In *Proceedings 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification*, 2001.

[19] C. Schulte. Programming deep concurrent constraint combinators. In Enrico Pontelli and Vtor Santos Costa, editors, *Second International Workshop on Practical Aspects of Declarative Languages*, volume 1753 of *LNCS*, pages 215–229. Springer-Verlag, 2000.

[20] INRIA Sophia-Antipolis, IRISA, AQL, and Oberthur. `http://www-sop.inria.fr/everest/projects/castles/`.

[21] JavaCard Technology. `http://java.sun.com/products/javacard`.

[22] CHR website. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`.